Cache-Aware Scheduling and Analysis for Multicores*

Nan Guan[1,2], Martin Stigge[1], Wang Yi[1,2] and Ge Yu[2]
[1]Department of Information Technology, Uppsala University, Sweden
[2]Department of Computer Science and Technology, Northeastern University, China
{nan.guan, martin.stigge, wang.yi}@it.uu.se, yuge@mail.neu.edu.cn

ABSTRACT

The major obstacle to use multicores for real-time applications is that we may not predict and provide any guarantee on real-time properties of embedded software on such platforms; the way of handling the on-chip shared resources such as L2 cache may have a significant impact on the timing predictability. In this paper, we propose to use cache space isolation techniques to avoid cache contention for hard real-time tasks running on multicores with shared caches. We present a scheduling strategy for real-time tasks with both timing and cache space constraints, which allows each task to use a fixed number of cache partitions, and makes sure that at any time a cache partition is occupied by at most one running task. In this way, the cache spaces of tasks are isolated at run-time.

As technical contributions, we have developed a sufficient schedulability test for non-preemptive fixed-priority scheduling for multicores with shared L2 cache, encoded as a linear programming problem. To improve the scalability of the test, we then present our second schedulability test of quadratic complexity, which is an over approximation of the first test. To evaluate the performance and scalability of our techniques, we use randomly generated task sets. Our experiments show that the first test which employs an LP solver can easily handle task sets with thousands of tasks in minutes using a desktop computer. It is also shown that the second test is comparable with the first one in terms of precision, but scales much better due to its low complexity, and is therefore a good candidate for efficient schedulability tests in the design loop for embedded systems or as an on-line test for admission control.

Categories and Subject Descriptors

C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS [**Real-time and embedded systems**]:

*This work was partially sponsored by ArtistDesign, CREDO, CoDeR-MP and UPMARC, and NSF of China under Grant No. 60773220.

EMSOFT'09, October 12–16, 2009, Grenoble, France.

General Terms

Design, Performance

Keywords

multicores, real-time systems, cache partitioning, schedulability analysis

1. INTRODUCTION

It is predicted that multicores will be increasingly used in future embedded systems for high performance and low energy consumption. The major obstacle is that we may not predict and provide any guarantee on real-time properties of embedded software on such platforms due to the on-chip shared resources. Shared caches such as L2 cache are among the most critical resources on multicores, which severely degrade the timing predictability of multicore systems due to the cache contention between cores.

For single processor systems, there are well-developed techniques [30] for timing analysis of embedded software. Using these techniques, the worst-case execution time (WCET) of real-time tasks may be estimated, and then used for system-level timing analyses like schedulability analysis. One major problem in WCET analysis is how to predict the cache behavior, since different cache behaviors (cache hit or miss) will result in different execution times of each instruction. The cache behavior modeling and analysis for single-processor architectures have been intensively studied in the past decades and are supported now in most existing WCET analysis tools [30]. Unfortunately the existing techniques for single processor platforms are not applicable for multicores with shared caches. The reason is that a task running on one core may evict the useful L2 cache content belonging to a task running on another core and therefore the Worst-Case Execution Time (WCET) of one task can not be estimated in isolation from the other tasks as for single processor systems. Essentially, the challenge is to model and predict the cache behavior for concurrent programs (not sequential programs as for the case of single processor systems) running on different cores.

To our best knowledge, the only known work on WCET analysis for multicores with shared cache is [32], which is only applicable to a special scenario and very simple hardware architecture (we will discuss its limitation in Section 2). Researchers in the WCET analysis community agree that "it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention among multiple cores in a shared cache" [27].

The goal of this paper is not to solve the above challenging problem. Instead, we use cache partitioning techniques such as page-coloring [8] combined with scheduling to isolate the cache spaces of hard real-time tasks running simultaneously to avoid the interference between them. This yields an efficient method – cache space isolation – to control the shared cache access, in which a portion of the shared cache is assigned to each running task, and the cache replacement is restricted to each individual partition. For single-processor multi-tasking systems, cache space isolation allows compositional timing analysis where the WCET of tasks can be estimated separately using existing WCET analysis techniques [11]. For multicores, to enable compositional timing analysis, we need isolation techniques for all the shared resources. For the on-chip shared bus bandwidth, techniques such as time-slicing, round-robin and prioritized access have been studied in e.g. [25, 24]. In this paper, we shall focus on shared caches only, and study the scheduling and analysis problem for hard real-time tasks with timing and cache space constraints, on multicores with shared L2 cache. We assume that the shared cache is divided into partitions, and further assume that the cache space size of each application task has been estimated by, for example, the miss-rate/cache-size curve or static analysis, and the WCET of each task is obtained with its assigned cache space size. In the system design phase, one can adjust tasks' L2 cache space sizes (and therefore their WCETs) to improve the system real-time performance, which can be built upon the schedulability analysis techniques studied in this paper.

We shall present a cache-aware scheduling algorithm which makes sure that at any time, any two running tasks' cache spaces are non-overlapped. A task can get to execute only if it gets an idle core as well as enough space (not necessarily continuous) on the shared cache. For the simplicity of presentation, we shall focus on non-preemptive fixed-priority scheduling. However, our results can be easily adapted to other scheduling strategies such as EDF, which is presented in the technical report version of this paper [20] . Our fist technical contribution is a sufficient schedulability test for multicores with shared L2 cache, encoded as a linear programming problem. To improve its scalability, we then propose our second schedulability test of quadratic complexity, which is an over approximation of the first test. To evaluate the performance and scalability of our techniques, we use randomly generated task sets. Our experiments show that the first test which employs an LP solver can easily handle task sets with thousands of tasks in minutes using a desktop computer. It is also shown that the second test is comparable with the first one in terms of precision, but scales much better due to its low complexity, and therefore it is a good candidate for efficient schedulability tests in the design loop for embedded systems.

The paper is structured as follows: Section 2 presents related work. Section 3 introduces the background of cache space isolation and the task model, and Section 4 introduces the scheduling algorithm $\mathsf{FP}_{\mathsf{CA}}$, as well as the analysis framework. The two schedulability tests for $\mathsf{FP}_{\mathsf{CA}}$ are presented in Section 5 and Section 6. Section 7 presents performance evaluation. Section 8 discusses extensions of the cache-aware scheduling, and finally, conclusions are given in Section 9.

2. RELATED WORK

Since L2 misses affect the system performance to a much greater extent than L1 misses or pipeline conflicts [17], the shared cache contention may dramatically degrade the system performance and predictability. Chandra et al. [13] showed that a thread's execution time may be up to 65% longer when it runs with a high-miss-rate co-runner than with a low-miss-rate co-runner. Such dramatic slowdowns were due to significant increases in L2 cache miss rates experienced with a high-miss-rate co-runner, as opposed to a low-miss-rate co-runner.

L2 contention can be reduced by discouraging threads with heavy memory-to-L2 traffic from being co-scheduled [17]. Anderson et al. [2, 1, 12] applied the policy of encouraging or discouraging the co-scheduling of tasks (or jobs), to improve the cache performance and also to meet the real-time constraints. All these works assumed that the WCETs of real-time threads are known in advance. However, although improved cache performance can directly reduce average execution costs, it is still unknown how to obtain the WCET of each real-time thread in their system model.

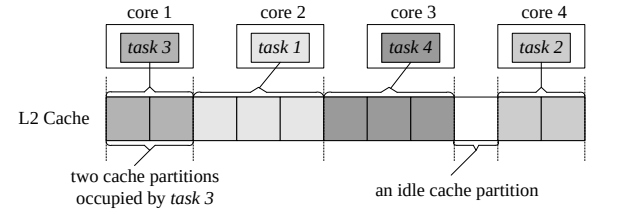
Yan and Zhang [32] is the only known work to studied the WCET analysis problem for multicore systems with shared L2 cache. A particular scenario is assumed that two tasks simultaneously run on a dual-core processor with a direct-mapped shared L2 instruction cache. However, their analysis technique is quite limited: firstly, most of today's multicore processors employ set-associative caches rather than direct-mapped cache as their L2 cache; secondly, when the system contains more cores and more tasks, their analysis will be extremely pessimistic; thirdly, their analysis technique can not handle tasks in priority-driven scheduling systems.

In contrast with Anderson's work, we employ cache space isolation in the scheduling algorithms in this paper, to avoid the cache accessing interference between tasks simultaneously running on different cores, and therefore we can apply existing analysis techniques to derive safe upper bounds of a task's WCET[1], with which we can do safe schedulability analysis for the task system.

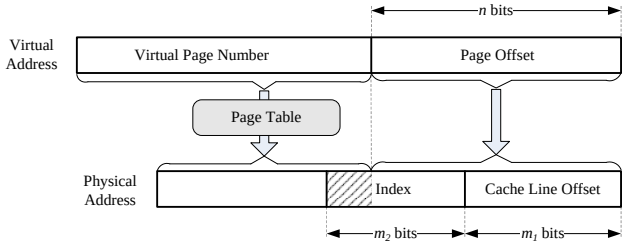
The schedulability analysis problem of global multiprocessor scheduling has been intensively studied [3, 4, 9, 6, 21, 22]. These analysis techniques are also extended to deal with more general cases, e.g., the global scheduling on 1-D FPGAs [15, 19], where a task may occupy multiple resources (columns on FPGAs) during execution. However, all these techniques are not applicable to our problem, since with cache space isolation, tasks are actually scheduled on two resources: cores and the shared cache.

Fisher et al. [18] studied the problem of static allocation of periodic tasks onto a multiprocessor platform such that on each processor, the total utilization of the allocated tasks is no larger than 1, as well as the total memory size of the allocated tasks does not exceed the processor's memory capacity. Suhendra et al. [28] and Salamy et al. [26] studied the problem of how to statically allocate and schedule a task graph onto a MPSoC, in which each processor

[1]In this paper we focus on the interference caused by the shared L2 cache, and there could be other interference between tasks running simultaneously. However, we believe the scheduling algorithm and analysis techniques in this paper is a necessary step towards completely avoiding interference between tasks running on multicores, and can be integrated with techniques of performance isolation on other shared resources, for instance, the work in [25, 24] to avoid interference caused by the shared on-chip bus.



(a) Cache Space Isolation



(b) Address mapping

Figure 1: Cache Space Isolation and Page Coloring

has a private scratch-pad memory, to maximize the system throughput. In summary, in the above work tasks are statically allocated to processors, so the schedulability analysis problem is trivial (reduced to the case of single processor scheduling). In our paper, different instances of a task are allowed to run on different cores, so the schedulability analysis problem is more difficult. In [27], several scheduling policies with shared cache partitioning and locking are experimentally evaluated, however, the schedulability analysis problem was not studied.

3. PRELIMINARIES

In this section, we briefly describe the basic assumptions on the hardware platform and application tasks, which our work is based on.

3.1 Cache Space Partitioning

We assume a multicore containing a fixed number of processor cores sharing an on-chip cache. Note that this is usually an L2 cache. We will not explicitly model core-local caches (usually L1) or other shared resources like interconnects. Since concurrent accesses to the shared cache give raise to the problem of reduced predictability due to cache interference, we assume the existence of a cache partitioning mechanism, allowing to divide the cache space into non-overlapping partitions for independent use by the computation tasks, see Figure 1(a).

Partitioning a cache shared among several tasks at the same time is a concept which has already been used, most notably, for reducing interference in order to improve average case performance or to increase predictability in single-core settings with preemption [31, 16, 10].

Different approaches may be used to achieve cache partitioning. Assuming a k-associative cache that consists of l cache sets with k cache lines each, one can distinguish *set-based* [8] and *associativity-based* [14] partitioning. The first one is also called *row-based partitioning* and assigns different cache sets to different partitions. It therefore enables up to l partitions and is thus quite fine-grained for bigger caches. The second one assigns a certain amount of lines within each cache set to different partitions and is also called *column-based partitioning*, so it is rather coarse-grained with a maximum of just k partitions. Mixtures of both variants are also possible. The approaches can be software- or hardware-based and differ regarding additional hardware requirements, partitioning granularity, influence on memory layout and the possibility as well as complexity of on-line repartitioning.

Here we give a brief description of a set-based approach, which is also known as *page coloring*. It has the advantage of being entirely software-based by exploiting the translation from virtual to physical memory addresses present in the virtual memory system[2]. Assume a simple hardware-indexed cache with cache line size of 2^{m_1} words and 2^{m_2} cache sets, so the least significant bits of the physical address will contain m_1 bits used as cache line offset and m_2 bits used as the set number, see Figure 1(b). Further assume a virtual page size of 2^n words, so the n least significant bits of the virtual address comprise the page offset. Consequently, all the other (more significant) bits are the page number and will be translated by the virtual memory system via the page table into the most significant bits of the physical address. If $m_1 + m_2 > n$ (which is the case with larger caches), a certain number of bits used to address the cache set are actually "controlled" by the virtual memory system, so that each virtual page can be (indirectly) mapped on a particular subset of all cache sets. The number of available *page colors* by that method is therefore $2^{(m_1+m_2)-n}$.

An example system supporting cache partitioning is reported in [29], where the authors modified the Linux kernel to support page-coloring based cache space isolation, in which 16 colors are supported, and conducted intensive experiments on a Power 5 dual-core processor. Note that the method enforces a certain (physical) memory layout, since it influences the choice of physical addresses. This restricts the memory size available to each task, as well as flexibility for recoloring. These problems can be compensated for by a simple rewiring trick as described in [23]. Therefore it is reasonable for our model to assume a cache with equally sized cache partitions that can be assigned and reassigned arbitrarily during the lifetimes of the tasks in question.

3.2 Task Model

Assume a multicore platform consisting of M cores and A cache partitions, and a set τ of independent sporadic tasks whose numbers of cache partitions (cache space size needed) and WCETs are known for the platform.

We use $\tau_i = \langle A_i, C_i, D_i, T_i \rangle$ to denote such a task where A_i is the *cache space size*, C_i is the *worst-case execution time* (WCET), $D_i \leq T_i$ is the relative deadline for each release, and T_i is the minimum inter-arrival separation time also referred to as the *period* of the task. We further assume that all tasks are ordered by priorities, i.e., τ_i has higher priority than τ_j iff $i < j$. The *utilization* of a task τ_i is $U_i = C_i/T_i$ and its *slack* $S_i = D_i - C_i$, which is the longest delay allowed before actually running without missing its deadline.

A sporadic task τ_i generates a potentially infinite sequence of *jobs* with successive job-arrivals separated by at least T_i

[2]Note that this is just an example of how cache partitioning can be achieved; by no means is virtual memory a necessity to the results presented in this paper.

Table 1: An example task set

Task	D_i	T_i	C_i	A_i
τ_1	3	3	2	1
τ_2	4	4	3	2
τ_3	5	5	2	2
τ_4	8	8	2	1

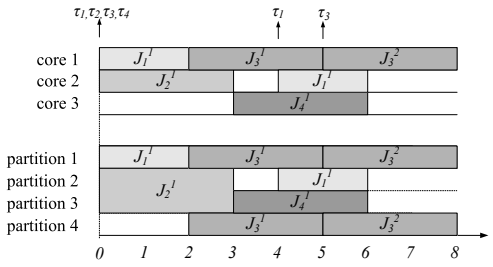


Figure 2: An example to illustrate $\mathsf{FP_{CA}}$.

time units. The α^{th} job of task τ_i is denoted by J_i^α, so we can denote a job sequence of task τ_i with $(J_i^1, J_i^2, \ldots)$. We omit α and just use J_i to denote a job of τ_i if there is no need to identify which job it is. Each job J_i adheres to the conditions A_i, C_i and D_i of its task τ_i and has additional properties concerning *absolute time points* related to its execution, which we denote with lower case letters: The *release time*, denoted by r_i, the *deadline*, denoted by d_i and derived using $d_i = r_i + D_i$, and the *latest start time*, denoted by l_i and derived using $l_i = r_i + S_i$. Finally, without loosing generality, we assume that time is dense in our model.

4. CACHE-AWARE SCHEDULING

We present the basic scheduling algorithm studied in this paper, and the analysis framework for the technical contributions presented in the next sections. We should point out that the simple scheduling algorithm itself is not the main contribution of this work. Our contributions are in solving the schedulability problem for this algorithm.

4.1 The Scheduling Algorithm $\mathsf{FP_{CA}}$

Since cache-related context-switch overhead of each task due to preemption is usually hard to predict, we focus in this paper on non-preemptive scheduling. The idea of cache space isolation can be applied to many different traditional multiprocessor scheduling algorithms, and for simplicity reasons, we will take the Non-preemptive Fixed Priority Scheduling as the example in this paper.

The algorithm, the Cache-Aware Non-preemptive Fixed Priority Scheduling ($\mathsf{FP_{CA}}$), is executed whenever a job finishes or when a new job arrives. It always schedules the highest priority waiting job for execution, if there are enough resources available. In particular, a job J_i is scheduled for execution if:

1. J_i is the job of highest priority among all waiting jobs,
2. There is at least one core idle, and
3. Enough cache partitions, i.e. at least A_i, are idle.

Note that, since we suppose $D_i \leq T_i$ for each task, there is at most one job of each task at any time instant.

Figure 2 shows an example of the task set in Table 1 scheduled by $\mathsf{FP_{CA}}$ (the scenario that all tasks are released together). Note that at time 0, the job J_4^1 can not execute according to the definition of $\mathsf{FP_{CA}}$, although it is ready and there is an idle core and enough idle cache partitions to fit it, since it is not at the first position of the waiting queue, i.e. there is a higher priority job (J_3^1) waiting for execution. J_3^1 can not execute since there is not enough idle cache partitions available. Thus, we note that $\mathsf{FP_{CA}}$ may *waste resources* as it does not schedule lower priority ready jobs to execute in advance of higher priority ready jobs even though there are enough resources available to accomodate them. However, it enforces a stricter priority ordering, which is in general good for predictability. We name this kind of scheduling policy as *blocking-style scheduling*.

Sometimes one may prefer to allow lower priority ready jobs to execute in advance of higher priority ready jobs, if the idle cache partitions are not enough to fit the higher priority ones, to trade predictability for better resource utilization. We name this kind of scheduling policy as *non-blocking-style scheduling*.

For simplicity reasons, we will present the schedulability analysis in context of $\mathsf{FP_{CA}}$, which is *blocking-style scheduling*. However, note that the schedulability analysis techniques are applicable to both *blocking-style scheduling* and *non-blocking-style scheduling*. Later in Section 8, we will discuss the comparison between them in more detail.

4.2 Problem Window Analysis

To check whether a given set of tasks can be scheduled using the above algorithm without missing the deadline for any job released, we shall study the time interval during which an assumed deadline missing task is prevented from running. Note that this interval is the so-called slack of the task, which we shall also call the *problem window* [4].

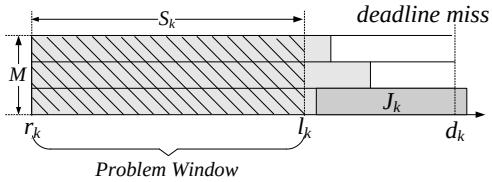
In the following, we outline how the problem window can be used for schedulability analysis in the case when tasks are scheduled only on the cores or only on the shared cache partitions. Two schedulability test conditions will be developed for the two special cases. Then, in Section 5, we combine them to deal with the general case.

4.2.1 The case without cache scheduling

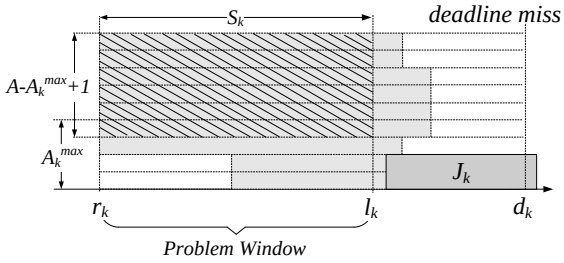
A schedulability test for the case when the tasks are scheduled only on the cores can be derived as follows:

1. Assume M cores for execution of a task set τ as described before, but in the task model, the A_i's are 0 (alternatively the total number of cache partitions is large enough such that no task will be blocked by a busy cache).

2. Suppose that the task set τ is unschedulable, then there is a job sequence $(J_{k_1}^{\alpha_1}, J_{k_2}^{\alpha_2}, \ldots)$ in which a job misses its deadline. Let J_k, a job of τ_k, be the first job missing its deadline. Its release time is r_k and the latest time point, at which it would have needed to start running (but it did not, since it is missing its deadline) is $l_k = r_k + S_k$. We define the time interval $[r_k, l_k]$ of length S_k as the *problem window*, as shown in Figure 3(a). The intuition is that at all time points within the interval, each of the cores must be occupied by another task, preventing J_k from running.



(a) Only considering scheduling on cores



(b) Only considering scheduling on the cache

Figure 3: Problem Window

3. To find out why J_k is not scheduled to run during the window, we may estimate the work load or an upper bound of this, generated by a task that may occupy a core in the window. We denote such an upper bound by I_k^i, which is normally called the interference contributed to J_k's problem window by task τ_i. The sum $\sum_i I_k^i$ is an upper bound of the total work load interfering with J_k in the problem window. We describe in detail how to calculate such an upper bound in the next section. A more precise calculation is given in the appendix.

4. We note that the non-preemptive fixed priority scheduling algorithm (without cache) enjoys the *work-conserving* property, that is, none of the M cores is idle if there is some ready job waiting for execution. Therefore, J_k can miss its deadline only if $\sum_i I_k^i \geq S_k \cdot M$ holds, i.e., the whole area with diagonals in Figure 3(a) is occupied. Otherwise, J_k is safe from ever missing its deadline, i.e., τ_k is schedulable, if the following condition holds: $\sum_i I_k^i < S_k \cdot M$.

We may also view this last step in a different way. We know that the sum of all work (of all tasks τ_i) interfering with J_k is bounded by $\sum_i I_k^i$, and it is in the worst case executed in parallel on M cores, thus preventing J_k from running. Therefore, if we divide this sum $\sum_i I_k^i$ by M, we get an upper bound on the maximum time that job J_k can be delayed by other tasks. We call this the *interference time*.

Consequently, J_k is guaranteed to be schedulable, if this interference time is strictly less than its slack, i.e., if the following condition holds:

$$\frac{1}{M}\sum_i I_k^i < S_k \tag{1}$$

By applying the above procedure to each task $\tau_k \in \tau$ (i.e., checking that the inequality holds for all tasks), one can construct a sufficient schedulability test for the case without a shared cache.

4.2.2 The case without core scheduling

The above problem window analysis can be generalized to the case where each task occupies *several* computing resources. In our scenario, one task can occupy *several* cache partitions at once while executing.

To present the idea, let us assume for the moment that we *only* care about the scheduling of the shared cache (suppose there are always enough cores for tasks to execute). A job J_i may start running as soon as it is the first one in the waiting queue Q_{wait}, and the number of idle cache partitions is at least A_i. Otherwise, J_k in Q_{wait} (it is now not necessarily the first job) may have to wait, if the number of idle cache partitions is less than $\max(A_1, \ldots, A_k)$.

Note that we take the maximum over all higher priority tasks here, since even though there might be A_k cache partitions idle, there could be a job J_i of higher priority ($i < k$) in Q_{wait} that needs more cache partitions to run, but is prevented from running, which in turn, prevents J_k from running, because of the blocking property of $\mathsf{FP_{CA}}$. We define this number as:

$$A_k^{\max} = \max_{i \leq k} A_i$$

Note that $A_k^{\max}$ is the minimal number of idle cache partitions needed in order that J_k is not blocked from running because of a busy cache. Equivalently, the minimal number of busy cache partitions that may block J_k from running, is $A - A_k^{\max} + 1$.

Therefore J_k can miss its deadline only if the whole area with diagonals in Figure 3(b) is occupied. Since each task τ_i is occupying A_i cache partitions while it is executing, we know that J_k can miss its deadline only if the condition $\sum_i A_i I_k^i \geq S_k \cdot (A - A_k^{\max} + 1)$ holds. Thus we have a test condition for scheduling analysis when the shared cache is considered: $\sum_i A_i I_k^i < S_k \cdot (A - A_k^{\max} + 1)$.

Like in Section 4.2.1, we again prefer the view on that in terms of *interference time*: We get an upper bound of the interference time suffered by job J_k in the problem window by dividing this sum of maximal total cache use $\sum_i A_i I_k^i$ by the minimal number of busy cache partitions $(A - A_k^{\max} + 1)$ throughout the problem window.

This is, again, an upper bound of the time, by which job J_k can be delayed by other tasks. Thus, the schedulability test condition is:

$$\frac{1}{A - A_k^{\max} + 1}\sum_i A_i I_k^i < S_k \tag{2}$$

As we see now in Constraints (1) and (2), one can derive test conditions for scheduling on cores and cache partitions *separately*, once I_k^i is known for each task τ_i. Since in the scheduling algorithm in question, $\mathsf{FP_{CA}}$, the scheduling happens on cores and cache *together*, the conditions have to be combined in a way that still makes for a safe schedulability test condition. In the following section, we will derive a novel way of combining both conditions.

5. THE FIRST TEST: LP-BASED

In order to apply the problem window analysis to $\mathsf{FP_{CA}}$, two questions need to be answered:

1. *How to compute I_k^i, i.e., an upper bound of the interference of each task τ_i in the problem window?* We will answer this question in Section 5.1.

2. *How to determine whether the interference of all tasks is large enough to prevent J_k from executing in the*

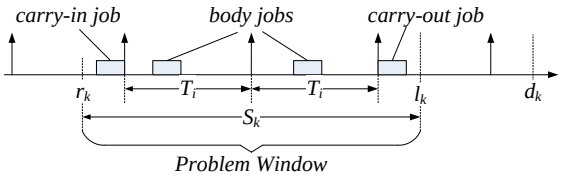


Figure 4: carry-in job, carry-out job and body jobs

problem window? We will answer this question in Section 5.2.

5.1 Interference Calculation

The first question can be answered by categorizing each job of τ_i in the problem window into one of three types, as shown in Figure 4:

body job: a job with both release time and deadline *in* the problem window; All the body jobs together contribute $\lfloor S_k/T_i \rfloor \cdot C_i$ to the interference.

carry-in job: a job with release time *earlier than* r_k, but with deadline *in* the problem window; This job contributes at most C_i to the interference.

carry-out job: a job with release time *in* the problem window, but with deadline *later* than l_k; This job also contributes at most C_i to the interference.

It follows that an upper bound of τ_i's interference in the problem window is given by

$$I_k^i = \left(\left\lfloor \frac{S_k}{T_i} \right\rfloor + 2 \right) \cdot C_i. \quad (3)$$

We can derive a more precise computation of I_k^i by carefully identifying the worst-case scenario of each task's interference, which is given in the appendix.

5.2 Schedulability Test as an LP Problem

The answer of the second question is non-trivial, and is the novel contribution of this paper.

As introduced in Section 4.2, the problem window analysis can be applied to analyzing the scheduling on cores or on the cache *separately*. However, if we consider the scheduling on both cores and cache, it is generally unknown what the lower bound of the occupied resources on each of them is, to cause J_k to miss deadline. For example, in Figure 2, at time instant 0, the job J_3^1 is ready for execution, but it can not execute since the number of idle cache partitions on the shared cache is not enough to accommodate it, so it is not true any longer that all M cores must be busy during the problem window to cause the considered task to miss its deadline.

5.2.1 Dividing up the problem window

The key observation to our analysis is, that at each time point in the problem window where the job J_k can not start running, all cores are already occupied, or – wherever that is not the case – enough cache partitions are occupied to prevent J_k from running. This is expressed in the following lemma about $\mathsf{FP_{CA}}$:

LEMMA 1. *Let J_k be a job that misses its deadline. Then at any time instant in the problem window $[r_k, l_k]$, at least one of the following two conditions is true:*

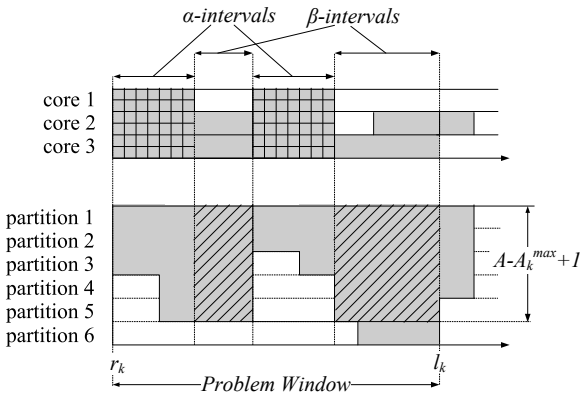


Figure 5: Illustration of α-intervals and β-intervals

1. *All M cores are occupied;*
2. *At least $A - A_k^{\max} + 1$ cache partitions are occupied.*

PROOF. Suppose there is a time instant $t \in [r_k, l_k]$, such that both of the above two conditions do not hold. Since J_k misses its deadline, it cannot start executing in the problem window, thus also not at t. Therefore, the waiting queue Q_{wait} is not empty at t, since at least J_k is in Q_{wait}.

Let now J_i be the first job in Q_{wait} at time t. Since, for $\mathsf{FP_{CA}}$, the waiting queue is ordered in strict priority order, J_i is the highest priority job waiting. By assumption, there are less than $A - A_k^{\max} + 1$ cache partitions occupied, so there are at least $A_k^{\max}$ partitions available. Further, $A_i \leq A_k^{\max}$ by definition, and there is an idle core by assumption. Thus, J_i must be able to execute, contradicting the assumption that it is waiting. □

Following these two conditions, we can now divide the problem window into two parts (see Figure 5):

1. *α-intervals*, in which all cores are busy;
2. *β-intervals*, in which at least one core is idle. Note that it follows from Lemma 1, that during the β-intervals, at least $A - A_k^{\max} + 1$ partitions of the shared cache are occupied by a running task.

It is generally unknown at what length of the α- and β-intervals the maximal interference to J_k is achieved. We approach this by introducing a Linear Programming (LP) formulation of our problem, to create a schedulability test for $\mathsf{FP_{CA}}$.

5.2.2 LP formulation

Suppose, as before, the task set τ is unschedulable by $\mathsf{FP_{CA}}$, and J_k is the first task that is missing its deadline. The time interval $[r_k, l_k]$ is the problem window.

The LP formulation will use the following constants:

- M: the number of cores.
- A: the total number of partitions on the shared cache.
- A_i: the number of cache partitions occupied by each task τ_i. (We also use the constant $A_k^{\max}$, which is derived from these as above.)

- I_k^i: an upper bound of the interference by τ_i in the problem window, which is computed as in Section 5.1 (or as in the appendix) for each τ_i.

Further, the following non-negative variables are used:

- α_i: for each task τ_i, we define α_i as τ_i's accumulated execution time during α-intervals.
- β_i: for each task τ_i, we define β_i as τ_i's accumulated execution time during β-intervals.

During the α-intervals, all M cores are occupied. Further, we know that $\sum_i \alpha_i$ equals to the total computation work of all tasks during the α-intervals (which is the area with grids in Figure 5). We can therefore express the accumulated length of all α-intervals as:

$$\frac{1}{M}\sum_i \alpha_i \tag{4}$$

During the β-intervals, at least $A - A_k^{\max} + 1$ cache partitions are occupied. Further, $\sum_i A_i\beta_i$ is the total cache partition use of all tasks during the β-intervals (which is an upper bound of the area with diagonals in Figure 5). We can therefore express an upper bound of the accumulated length of all β-intervals as:

$$\frac{1}{A - A_k^{\max} + 1}\sum_i A_i\beta_i \tag{5}$$

Since J_k is not schedulable, we further know that the sum of the accumulated lengths of the α- and β-intervals is at least S_k. Thus, using the expressions from (4) and (5), it must hold that:

$$\sum_i \left(\frac{1}{M}\alpha_i + \frac{A_i}{A - A_k^{\max} + 1}\beta_i\right) \geq S_k \tag{6}$$

We can use an LP solver to detect whether the α_i and β_i variables can be chosen in a way to satisfy this condition. If this is not the case, then τ_k would be schedulable. We can use the object function of our LP formulation for that check:

$$Maximize \quad \sum_i \left(\frac{1}{M}\alpha_i + \frac{A_i}{A - A_k^{\max} + 1}\beta_i\right) \tag{7}$$

Thus, if the solution of the LP problem is smaller than S_k, we can determine that τ_k is schedulable.

So far, the variables α_i and β_i are not bounded, so without further constraints, the LP formulation will not have a bounded solution (which would trivially render all tasks unschedulable). Therefore, we add constraints on the free variables, that follow directly from the structure of our schedulability problem. We have three constraints:

φ_1: **Interference Constraint** We know that I_k^j is the upper bound of the work done by τ_j in the problem window, so we have:

$$\forall j : \alpha_j + \beta_j \leq I_k^j$$

φ_2: **Core Constraint** The work done by a task in the α-intervals can not be larger than the total accumulated length of the α-intervals (see Expression (4)), so we have:

$$\forall j : \alpha_j \leq \frac{1}{M}\sum_i \alpha_i$$

φ_3: **Cache Constraint** The work done by a task in the β-intervals can not be larger than the total accumulated length of the β-intervals. Thus, it can not be larger than the upper bound of the total length of the β-intervals (see Expression (5)), so we have:

$$\forall j : \beta_j \leq \frac{1}{A - A_k^{\max} + 1}\sum_i A_i\beta_i$$

To test τ_k for schedulability, we can now invoke an LP solver on the LP problem defined by constraints φ_1 to φ_3 and the object function in (7). By construction, we have a first schedulability test for τ:

THEOREM 1 (THE FIRST TEST). *For each task τ_k, let χ_k denote the solution of the LP problem shown above. A task set τ is schedulable by $\mathsf{FP_{CA}}$, if for each task $\tau_k \in \tau$ it holds that*

$$\chi_k < S_k. \tag{8}$$

6. THE SECOND TEST: CLOSED FORM

Although the LP-based test presented in the previous section exhibits quite good scalability (as will be shown in Section 7), simple test conditions are often preferred, in e.g., online admission control and efficient analysis in the systems design loop. Thus, we will present a second schedulability test, which can be seen as an over-approximation of the LP-based test. It has quadratic computational complexity.

In the LP-based test, each task τ_i finds its interference I_k^i divided into the two parts α_i and β_i, expressed by constraint φ_1. From the object function (7) one can see that if $1/M \geq A_i/(A - A_k^{\max}+1)$, τ_i tends to contribute with as much α_i as possible, as long as φ_2 is respected; likewise in the opposite case with β_i. However, since the accumulated lengths of the α- and β-intervals (as used on the right hand sides of φ_2 and φ_3) are also dependent on the unknown variables, it is in general unknown how the α_i and β_i variables are chosen to maximize the interference time caused by all tasks. Therefore, the first schedulability test employs the LP solver to aid in "the search over all possible cases" for the maximal solution.

However, if we take out the constraints φ_2 and φ_3 from the LP formulation, each task τ_i will contribute $\alpha_i = I_k^i, \beta_i = 0$ if $1/M \geq A_i/(A - A_k^{\max} + 1)$, or $\beta_i = I_k^i, \alpha_i = 0$ otherwise, to maximize the object function. With this observation, we can derive a closed-form schedulability test which does not need to solve a search problem:

THEOREM 2 (THE SECOND TEST). *For each task τ_k let*

$$\chi_k^* := \sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot I_k^i.$$

A task set τ is schedulable by $\mathsf{FP_{CA}}$, if for each task $\tau_k \in \tau$ it holds that

$$\chi_k^* < S_k. \tag{9}$$

PROOF. We prove the theorem indirectly. Let τ be a task set not schedulable by $\mathsf{FP_{CA}}$, and J_k the deadline missing task as before. We already know that this implies the existence of a solution for the LP problem, such that in particular, φ_1 to φ_3 hold, and the value of the object function satisfies the following inequality:

$$\sum_i \left(\frac{1}{M} \cdot \alpha_i + \frac{A_i}{A - A_k^{\max} + 1} \cdot \beta_i\right) \geq S_k$$

M	$A - A_k^{\max} + 1$	I_k^1	A_1	I_k^2	A_2	I_k^3	A_3
2	4	4	1	4	3	6	1

Table 2: An example task set.

By relaxing the inequality, we get:

$$\sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot (\alpha_i + \beta_i) \geq S_k$$

Now, we apply condition φ_1:

$$\underbrace{\sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot I_k^i}_{\chi_k^*} \geq S_k$$

The theorem follows. □

Note that the upper bound χ_k^* derived in the above theorem is an over-approximation of the LP solution χ_k in the previous section.

For example, consider a task set with the interference parameters as stated in Table 2. The LP problem (from the first test) has the following solution:

$$\begin{array}{ll} \alpha_1 = 4 & \beta_1 = 0 \\ \alpha_2 = 1 & \beta_2 = 3 \\ \alpha_3 = 3 & \beta_3 = 3 \end{array}$$

This results in an upper bound of $\chi_k = 7$, which is the value of the object function.

In the second test, for χ_k^*, each task τ_i contributes all I_k^i as α_i if $1/M > A_i/(A - A_k^{\max} + 1)$, and as β_i otherwise, so we get the following bound:

$$\chi_k^* = \frac{1}{2} \cdot 4 + \frac{3}{4} \cdot 4 + \frac{1}{2} \cdot 6 = 8$$

Although the simple test condition is more pessimistic than the LP-based test, we found by extensive experiments, that the performance of the second test is very close to the LP-based test in terms of acceptance ratio. We will show that in Section 7. It follows that, for practical matters, the second test does not lose much precision for most task sets, while being of comparatively low complexity (quadratic with respect to the number of tasks).

Note that both the first and second schedulability test are sustainable, which means that a task set determined as schedulable by the test is still schedulable if the workload of some task is reduced (C_i is decreased or T_i is increased). The proof is omitted due to the page limitation.

7. PERFORMANCE EVALUATION

At first we evaluate the performance of the proposed schedulability tests in terms of acceptance ratio. We follow the method in [5] to generate task sets: A task set of $M + 1$ tasks is generated and tested. Then we iteratively increase the number of tasks by 1 to generate a new task set, and all the schedulability tests are run on the new task set. This process is iterated until the total processor utilization exceeds M. The whole procedure is then repeated, starting with a new task set of $M + 1$ tasks, until a reasonable sample space has been generated and tested. This method of generating random task sets produces a fairly uniform distribution of total utilizations, except at the extreme end of low utilization.

Table 3: Running time and peak memory usage of lpsolve to solve the LP formulation in the first test

Number of Tasks	4000	6000	8000	10000
Time in LP (s)	49.24	114.53	208.45	334.95
Mem. in LP (KB)	20344	28876	37556	46664

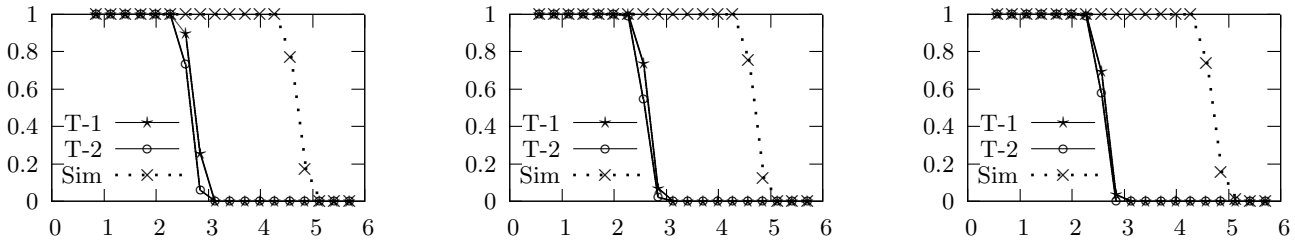
Figure 6 shows the acceptance ratio of the first test (denoted by "T-1"), and second test (denoted by "T-2") and the simulation (denoted by "Sim"). Since it is not computationally feasible to try all possible task release offsets and inter-release separations exhaustively in simulations, all task release offsets are set to be zero and all tasks are released periodically, and simulation is run for the hyper-period of all task periods. Simulation results obtained under this assumption may sometimes determine a task set to be schedulable even though it is not, but they can serve as a coarse upper bound of the acceptance ratio.

The parameter setting in Figure 6(a) is as follows: the number of cores is 6; the number of cache partitions is 40; for each task τ_i, T_i is uniformly distributed in $[10, 20]$, U_i is uniformly distributed in [0.1, 0.3] and A_i is uniformly distributed in $[1, 5]$, and we set $Di = Ti$. We can see that the performance of the first test is a little better than the second test. In Figure 6(b), the range of U_i is changed to $[0.1, 0.6]$ and other settings are the same as Figure 6(a). In Figure 6(c), the range of A_i is changed to $[2, 10]$ and other settings are the same as Figure 6(a). We can see that, in both cases, the acceptance ratio of all the simulations and tests degrades a little bit when the average utilization of tasks is slightly decreased, and the difference between the performance of the first test and second test is even smaller. In summary we can see that the second test does not lose too much precision, compared to the first test condition.

As mentioned earlier, the second test is of $O(N^2)$ complexity. The scalability of the first test is of our special concerns since it employs the LP formulation. We use the open source LP solver lpsolve [7] to solve the LP formulation of the first test. Table 3 shows the running time and maximal peak memory usage of lpsolve with different task set scales. The experiment is conducted on a normal desktop computer with an Intel Core2 processor (2.83GHz) and 2G memory. The experiments show that the first test can handle task sets with thousands of tasks in minutes.

8. BLOCKING VS. NON-BLOCKING

As we mentioned in Section 4.1, $\mathsf{FP_{CA}}$ may introduce a type of resource wasting in certain situations, caused by a difference in tasks cache requirements, in combination with strict adherence to priority ordering. The possible scenario is that when the current idle cache is not enough to fit the first job in the waiting queue (the highest-priority job among all ready tasks), there could be some lower-priority job in the waiting queue with a fitting cache requirement. In $\mathsf{FP_{CA}}$ as analyzed so far, lower-priority waiting jobs are not allowed to start in advance of the first job in the waiting queue since the priority ordering is enforced strictly. We call this sort of policy the *blocking-style* scheduling. The blocking-style scheduling would waste computing resources to guarantee the execution order of waiting jobs, as we saw earlier in the example in Figure 2. But it will not suffer from the unbounded priority inversion problem due to the sharing of



(a) $0.1 \leq U_i \leq 0.3, 1 \leq A_i \leq 5$ (b) $0.1 \leq U_i \leq 0.6, 1 \leq A_i \leq 5$ (c) $0.1 \leq U_i \leq 0.3, 2 \leq A_i \leq 10$

Figure 6: Acceptance Ratio: X-axis is total utilization $\sum_i U_i$; Y-axis is acceptance ratio.

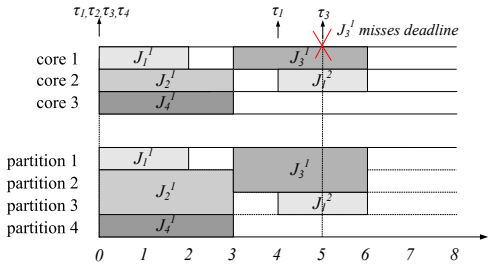


Figure 7: The non-blocking version of $\mathsf{FP_{CA}}$.

cache partitions as for the non-blocking approach described below.

Alternatively, to improve resource utilization, a lower-priority waiting job may be allowed to start execution in advance of the first job in the waiting queue, if the above described situation occurs, which we call *non-blocking-style* scheduling. Figure 7 shows how the task set in Table 1 is scheduled by the non-blocking-style version of $\mathsf{FP_{CA}}$. In this variant, the scheduler always runs the highest priority waiting job in the queue among all jobs that can actually run, given their resource (i.e. cache) constraints. This is done until there are no more jobs of that kind. In the example in Figure 7, we can see that at time instant 0, job J_4^1 starts execution although J_3^1 can not start.

From the predictability point of view, the blocking-style scheduling is usually to be preferred, in which waiting jobs start execution in strict priority orders. The reason is that τ_h may suffer more interference than it is the case in the blocking-style scheduling, since in non-blocking-style scheduling, a lower priority task τ_l can execute earlier than a higher priority task τ_h, and must run to completion because of non-preemptive scheduling. As shown in Figure 7, due to the advanced execution of J_4^1, J_3^1's start time is delayed to time 3, and J_3^1 will finally miss its deadline. In the worst case, this priority inversion effect could even cause unbounded interference to a task with even the highest priority.

On the other hand, the non-blocking-style scheduling utilizes resources better, since it always tries to utilize the computing resources as much as possible. Regarding the complexity at runtime, this comes with the cost that the scheduler needs to keep track of more than the head of the priority queue, since lower priority tasks might be able to run. The blocking-style variant is more lightweight, since only the head of the priority queue needs to be checked.

The system designer can choose to use blocking-style or non-blocking-style scheduling, as well as some compromise policy as a mixture of these two alternatives, according to the application requirement. It is out of the scope of this paper to compare them in detail. However, even though the schedulability analysis techniques proposed in this paper are done in the context of the blocking-style scheduling, they are also applicable to the non-blocking-style scheduling. For this, it is only necessary to incorporate some extra consideration of the interference caused by the lower priority jobs that may execute in advance of the analyzed job. We omit the detailed presentation of the analysis of non-blocking-style scheduling here due to page limitations, and referred the interested readers to our technical report [20].

9. CONCLUSIONS

The broad introduction of multicores brings us many interesting research challenges for embedded systems design. One of these is to predict the timing properties of embedded software on such platforms. One of the main obstacles is the sharing of on-chip caches such as L2. The message of this paper is that with proper resource isolation, it is possible to perform system-level schedulability analysis for multicore systems based on task-level timing analysis using existing WCET analysis techniques. Our contributions include two efficient techniques for such analyses in the presence of a shared cache. We may argue that hard real-time applications should be placed in local caches such as L1. An interesting future work is to develop techniques for estimating the cache space requirements of tasks. However, when there is not enough local cache space, the techniques presented here will be needed. We believe that our analysis techniques are also applicable to handle other types of on-chip resources such as bus bandwidth. We leave this for future work. As future work, we will also study how the allocation of cache space size for individual tasks will influence system-level performance and timing properties.

10. REFERENCES

[1] J. Anderson and J. Calandrino. Parallel real-time task scheduling on multicore platforms. *RTSS*, 2006.

[2] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. *RTAS*, 2006.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *RTSS*, 2001.

[4] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *RTSS*, 2003.

[5] T. P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. In *Technical Report, Florida State University*, 2005.

[6] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, 2007.

[7] M. Berkelaar. lp_solve: a mixed integer linear program solver. In *Relatorio Tecnico, Eindhoven University of Technology*, 1999.

[8] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding conflict misses dynamically in large direct mapped caches. In *ASPLOS*, 1994.

[9] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *ECRTS*, 2005.

[10] B.D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*, 2008.

[11] J. Vicente Busquets-Mataix and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *ECRTS*, 1997.

[12] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *ECRTS*, 2008.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a multi-processor architecture. In *HPCA*, 2005.

[14] D. Chiou, S. Devadas, L. Rudolph, and B. S. Ang. Dynamic cache partitioning via columnization. In *Technical Report, MIT*, 1999.

[15] K. Danne and M. Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. In *LCTES*, 2006.

[16] S. Dropsho and C. Weems. Comparing caching techniques for multitasking real-time systems. In *Technical Report, University of Massachusetts-Amherst*, 1997.

[17] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Throughput-oriented scheduling on chip multithreading systems. *Technical Report, Harvard University*, 2005.

[18] N. Fisher, J. Anderson, and S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *RTCSA*, 2005.

[19] N. Guan, Q. Deng, Z. Gu, W. Xu, and G. Yu. Schedulability analysis of preemptive and non-preemptive edf on partial runtime-reconfigurable fpgas. In *ACM Transaction on Design Automation of Electronic Systems*, volume 13, 2008.

[20] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Technical Report, Uppsala University, (http://user.it.uu.se/~yi)*, 2009.

[21] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *RTSS*, 2008.

[22] H. Leontyev and J. Anderson. A unified hard/soft real-time schedulability test for global edf multiprocessor scheduling. In *RTSS*, 2008.

[23] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *RTAS*, 1997.

[24] M. Paolieri, E. Quinones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, 2009.

[25] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[26] H. Salamy and J. Ramanujam. A framework for task scheduling and memory partitioning for multi-processor system-on-chip. In *HiPEAC*, 2009.

[27] V. Suhendra and T. Mitra. Exploring locking and partitioning for predictable shared caches on multi-cores. In *DAC*, 2008.

[28] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *CASES*, 2006.

[29] D. Tam, R. Azimi, and M. Stumm L. Soares. Managing shared l2 caches on multicore systems in software. In *WIOSCA*, 2007.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. In *ACM Transaction on Embedded Computing Systems*, 2008.

[31] A. Wolfe. Software-based cache partitioning for real-time applications. In *Journal of Computer Software Engineering*, 1994.

[32] J. Yan and W. Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS*, 2008.

Appendix

Improving the Interference Computation

The computation of I_k^i, an upper bound of the interference caused by τ_i over J_k, in Equation 3 (Section 5.1) is grossly over pessimistic. In the following we will present a more precise computation of I_k^i by carefully identifying the worst-case scenario of τ_i's interference.

Recall that the problem window $[r_k, l_k]$ is a time frame of a given length ($l_k - r_k = S_k$) for which we want to derive a bound of how much interference a task τ_i (or rather its jobs) can cause to possibly prevent J_k from running. We can compute I_k^i using the following lemma (proof in [20]):

LEMMA 2. *An upper bound of the interference by τ_i in the problem window of length S_k can be computed by:*

$$I_k^i = \begin{cases} S_k & i < k \wedge S_k < C_i \\ \left\lfloor \frac{S_k - C_i}{T_i} \right\rfloor C_i + C_i + \omega & i < k \wedge S_k \geq C_i \\ 0 & i = k \\ \min(C_i, S_k) & i > k \end{cases} \quad (10)$$

where $\omega = \min\Big(C_i, \max\big(0, (S_k - C_i) \bmod T_i - (T_i - D_i)\big)\Big)$.